## A View on Components

Lassing, N.; Rijsenbrij, D.; Vliet, H.

***published in***
Proceedings of the 9th International Workshop on Database and Expert Systems Applications (DEXA98)
1998

***document version***
Publisher's PDF, also known as Version of record

# A View on Components

N.H. Lassing
*Vrije Universiteit, Amsterdam*
*nlassing@cs.vu.nl*

D.B.B. Rijsenbrij
*Vrije Universiteit, Amsterdam*
*daan@cs.vu.nl*

J.C. van Vliet
*Vrije Universiteit, Amsterdam*
*hans@cs.vu.nl*

## Abstract

*Components are nowadays considered the next step in information system development. Components are assumed to foster reuse and flexibility, and reduce the complexity of distributed deployment. The purpose of this paper is to investigate the properties of components that determine whether the above goals are met. To that end, we explored the literature and had a number of interviews with representatives from tool-vendors, tool-users and software houses. The resulting views are summarized in this paper, and applied to a small example. In our further research, the architecture sketched in this example will be worked out in further detail, and compared with the architecture of similar systems found in industry. Such will deepen our understanding and assessment of architectural choices made.*

## 1. Introduction

Many organizations regard components as the next step in information system development. This is illustrated by the increasing number of vendors that are selling tools to support component-based development (CBD). These tools enable the developer to build components, combine components and deploy components. However, not every component that is built with such a tool, fully exploits the benefits of CBD. Some of the components we build are better than others. We would like to gain insight into the properties of a component that determine whether one component is better than another. The criterion to judge whether one component is better than another is its suitability to support the goals of the use of components.

To ensure that our results are applicable in concrete situations we explored the existing views on components. This exploration included a study of the literature and interviews with companies investing in components. These included tool-vendors, tool-users and software houses. They all have their own view on components. The purpose of this paper is to summarize the ideas behind these views.

In section 2 we indicate the goals we like to achieve by using components. In section 3 we describe the existing ideas about components. In section 4 we describe the relation between components and frameworks, two concepts that are often used in combination. And in section 5 we apply the ideas of the first sections in an example.

## 2. Goals

We can distinguish a number of goals we would like to achieve by using components. The most important of these are:

- Reuse (or multiple-usage as some people call it)
- Flexibility
- Reduction of the complexity of distributed deployment

For each of these goals we will discuss the benefits we expect to gain and the challenges we face when we try to achieve them.

### 2.1. Reuse

With the rise of components, the attention for reuse also increased, for it is believed that components are the optimal unit for reuse. Reuse is expected to have a number of potential benefits. The most important of these are a reduction of the time-to-delivery, a reduction of the development costs and a reduction of the number of errors in the delivered information system ([6]).

We have been trying to achieve reuse for three decades now, but we have not been very successful so far. This is caused by a number of problems. The main problem is that it is hard to determine which functionality a component should possess to be optimally reusable. Van Vliet ([10]) states that “a reusable component is to be valued, not for the trivial reason that it offers relief from implementing the functionality yourself, but for offering a piece of the right domain knowledge, the very functionality you need, gained through much experience and an obsessive desire to find the right abstractions”.

This means that our goal should not be to find components that are usable in every domain, but that it is more important to find the right components for a certain domain.

A more technical objection to reuse is what Garlan et al. call architectural mismatch (see [5]). Architectural mismatch occurs when the assumptions a component and its environment make about each other are conflicting. This complicates the collaboration of a collection of components. To prevent architectural mismatch a number of things need to be in harmony. First, the expectations the environment has of a component should agree with the actual service provided by that component (for example performance). And second, the assumptions a component makes about its environment should agree with the actual environment. The component could, for example, assume the presence of an event-handler in the environment, without which the component will not function.

Two approaches to the reduction of architectural mismatch can be distinguished. The first one is to make the service provided by the component and the expectations the component has of its environment explicit. This can be captured in a contract. We will describe a possible form of such a contract in section 3.2.

The second approach is the use of software architectures. Shaw and Garlan state that software architecture involves the description of elements from which information systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns ([8]). Thus, a software architecture indicates which types of components are used in an information system and how they collaborate. It puts constraints on the components, thereby decreasing the risk of architectural mismatch.

To implement the concepts described in a software architecture we can build a framework. In section 4 we will describe the relation between components and frameworks.

### 2.2. Flexibility

Flexibility can be described as the ease with which an application can be adapted to changing requirements. The flexibility of a system is increased if changes to the requirements impact only a limited part of that system. Therefore, we would like to subdivide our system into a number of components that can be changed (within certain limits) without affecting other components. We can be certain that a change to a component does not affect other components if the contract of the component does not change.

We now have two challenges for the design of the system. First we need to divide the system into components in such a way that changes in the requirements impact as few components as possible. Such a division can only be found if we have insight in the changes that can be expected. And second we need to find components that can be changed without affecting other components. The contracts of these components should leave room for certain types of changes.

### 2.3. Reduction of the complexity of distributed deployment

An increasing number of information systems is divided into a number of elements which are distributed over several clients and servers. The distribution of the elements impacts the performance, stability and security of the information system. In finding the optimal distribution of the information system the developer has to consider the following factors:

1. the expectations that each distribution element has of its environment,
2. relations between elements that determine whether these elements have to be put on the same location,
3. the required performance, stability and security.

The first two factors limit the freedom of the developer, they are preconditions for the distribution of elements. The task of the developer is to find a distribution that satisfies these preconditions and yet delivers the required performance, stability and security. When information systems consist of a large number of elements, this task can get very complicated.

Using components can reduce this complexity, if the components satisfy two conditions. The first condition is that the expectations a component has of its environment are described in a contract. The second condition is that components can be distributed independently of each other, i.e. the placement of component A on location L does not imply that component B is placed on location L too.

## 3. Components

A number of different views on components exist. In this section we describe the common ideas used in these various views.

### 3.1. Definition of a component

The term component is used for a number of concepts. If we accumulate the common elements of the different views on components, we get the following definition:

> *A component is a part of a system that can be separated from the rest of the system.*

However, this definition is very general and gives little to go on. By using components that satisfy this definition we do not automatically achieve the goals mentioned in section 2. To support these goals components should at least have the following additional properties:

- The component should represent a recognizable concept for the developer
- The component only communicates through interfaces defined for the component
- It is clearly defined what can be expected from the component and what the component expects from its environment
- The component can be distributed independently of other components.

This list is probably not yet complete; during further research it will be expanded with other properties.

### 3.2. Contracts

In the previous sections we mentioned that the services of a component should be made explicit in a contract. A contract of a component should include a specification of the following aspects:

- Functionality: a specification of the function of the component.
- Environment: a specification of the environment that the component expects. This includes a specification of the infrastructure, the control-structure and the data model the component expects in its environment (see [5]).
- Interfaces: a specification of the way the component can be invoked as well as a specification of how the component calls other components. This does not only include function names, but also the structure of parameters given and protocols used.
- Service level: a specification of the service levels delivered by the component. This includes things like performance, reliability, and scalability. How these levels can be measured and specified is not yet clear. This is a topic of further research.

### 3.3. Types of components

As mentioned in section 2.1 the software architecture of a system indicates the types of components that are used in a system. We argue that it is useful to classify the various types of components that can be distinguished into a number of categories. If we have a number of categories of components and the characteristics of these categories are considered when the software architecture is formulated, the chance that existing components can be reused is increased.

We classify components into two categories: IT-components and business components. IT-components are technical in nature and represent concepts that are mainly used by developers. Business components represent concepts from the user-domain. For these categories we encountered various types of components in our sessions with industry.

**IT-components**:

- *user interface-components*: components that implement elements of the user interface. This includes things like buttons, list-boxes and edit-boxes.
- *technical infrastructure components*: components that offer technical services that are used by other components of the system. This includes things like an event-handler, a database-management system and an authorization manager.

**Business components**:

- *functional components*: components that perform some sort of function. An example of this is an interest calculation-component.
- *business objects*: components that implement the data and functions of a concept from the business domain. The state of these objects should be persistent over the life of the application. Examples of this are a customer-component and an order-component.
- *collaboration components*: components that implement a (template) collaboration between a number of components. In [2] these components are called frameworks. An example of this is a sales-component, that is a collaboration between an order- and a customer component.

### 3.4. Approaches to components

We encountered two different approaches of tools to components. The first approach is to see components as a piece of executable software, together with a specification of its interface, and the second is to see a component as a fragment of a model. The major difference between these approaches is that in the first approach components are only executable on a specific platform, while in the second approach an implementation of the components can be generated for various execution platforms.

The major advantages of generating the implementation from models are that the components are not platform-specific and they can be easily adapted. The major disadvantage, however, is that the models can only be used in the tool they are defined in, although this is changing by developments like UML and tool-independent languages. Executable components, on the other hand, can sometimes be used in different tools, but they are platform-specific and not easily adaptable.

## 4. Components and frameworks

Frameworks are often regarded as an environment for the reuse of components ([3] and [7]). Therefore, it is useful to explore how components and frameworks are related.

A framework consists of a number of concrete and abstract classes and a definition of the way instances of these classes collaborate. The concrete classes implement the collaboration of the classes. The abstract classes are the points at which the framework can be used or adapted.

A framework usually implements some technical mechanism, but the same ideas can be applied to business mechanisms. A framework implements the larger part of the complexity of these mechanisms. The technical mechanisms include things like a user interface, object persistency and communication between distributed components. The business mechanisms include things like order-administration and logistics.

An application can use the framework in two different ways ([9]). The first is to inherit from the abstract classes provided by the framework. The second way is to use the services provided by the framework by calling functions defined in its interface.

Abstract classes not only serve as the base of the classes of an application built on the framework, but are also used as variation points of a framework. A variation point is a part of the framework for which the implementation is not given, but only the specification of the interface. These points allow the framework to be adapted to a specific situation. Variation points do, however, need to be filled in with a concrete class that implements their interface, otherwise the framework will not function. An example of this is a framework for object-persistency in which the persistency-broker itself is left open as a variation point. The framework implements the collaboration between the entities of the framework and the persistency-broker. However, only the definition of the interface of the persistency-broker is given, possibly together with a number of options for its implementation. The developer can now choose which implementation for the persistency-broker he prefers to use.

A framework is derived from a software architecture. The architecture indicates which components are used in the system and how they interact, and the framework provides an environment in which these components can be placed.

## 5. Example: Personnel Administration System

In this section we show how these ideas can be applied in a (simple) application. The example concerns a personnel administration system. Personnel administration systems are used in different domains. They share a common core of functionality that is extended with domain-specific functionality. However, we have decided to keep things rather simple and focus on common functionality.

### 5.1. Required functionality

The conceptual model of the domain under study is shown in the class diagram[1] of Figure 1. The system that we use in this example records information about the concepts shown in this diagram.

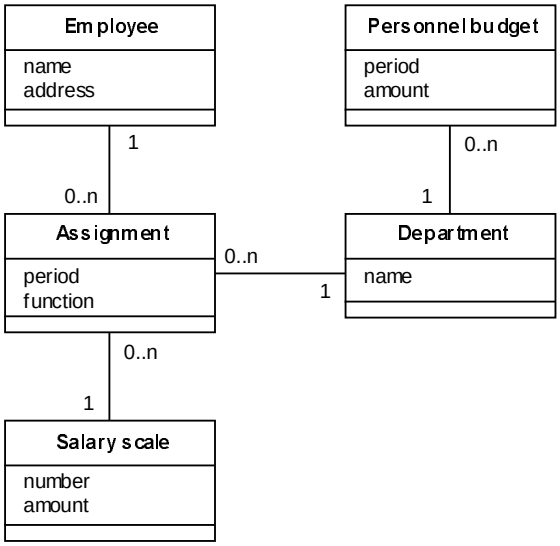


**Figure 1. Class diagram of personnel administration**

The system should support functions like:

- Create/delete/edit employee
- Create/delete/edit department
- Create/delete/edit salary scale
- Assign an employee to a department
- Delete assignment
- Assign personnel budget to a department
- Report comparison personnel budget and personnel cost per department
- Report employees per department

### 5.2. Software architecture

In this section we outline the software architecture for the personnel administration system. During the development of this software architecture we consider the desired level of flexibility and distribution.

For the distribution of the application we had to make a number of choices. We have decided that the application can be accessed from a number of distributed clients. As a corollary, the user interface is replicated for

[1] All class diagrams are depicted in the UML notation (see [4]).

each client. Then we had to choose whether the data of the application is replicated on different servers or not. We chose not to do so. For reasons of security the data needs to be located on a secure server and we presuppose only one server that is placed in a secure environment. And finally we had to choose where the processing of the application should take place: on the clients, on a server or on multiple servers. This question is left unanswered and we demand that our software architecture enables us to decide later where the processing should take place.

Thus, from the point of view of distribution we need three separate layers: a user interface-layer, a processing-layer and a data management-layer. The user interface-layer and the data management-layer in their entirety can be used as units of distribution. The user interface-layer is replicated for each client and the data management-layer is located on our secure server. The processing-layer should be divided into smaller units of distribution, because this layer can be distributed over multiple locations. We chose to create a unit of distribution for each function the application can perform. Figure 2 shows the layers and units of distribution of the application.

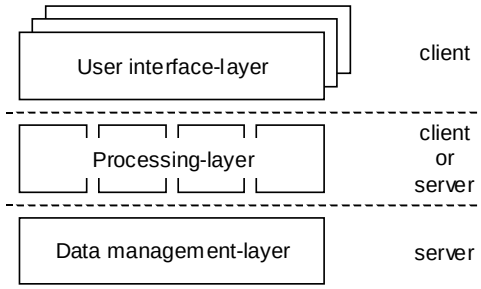


**Figure 2. The layers of the application and their location**

We indicated that the flexibility of an application is increased if changes to the requirements only impact a small number of elements of our application and changes to an element impact as few other components as possible. To limit the impact of changes in the requirements we should consider the expected changes to the requirements. In our example we expect that in the future the system will also be used to record additional information about the employee, such as career paths and courses taken. These changes necessitate an extension of the class diagram and the addition of a number of functions. We anticipate these changes to the class diagram by dividing the data management-layer into a component for each class of the class diagram, which we will call business objects. This way, when we add an class to the class diagram, we just have to add another business object and existing business objects remain the same. Because we have already divided the processing-layer into components for each function we can add function-components without the need to change existing function-components. The user interface-layer consists of the windows and dialog-boxes that present information to the user and receive input from the user. To enhance the flexibility these windows are also created according to the class diagram. Thus, for each business object and each relation a separate window is created.

To limit the impact of changes to a component we have chosen to put a number of constraints on these layers. The first constraint is that all access to a component goes through the interface defined in its contract and the second is that components of a layer are only accessed by components of the next higher layer. The advantage of this approach is that if changes to a component do not impact the contract, components of other layers do not have to be adapted and, if the contract should change, only components of higher layers need to be adapted. Figure 3 shows the subdivision of our application.

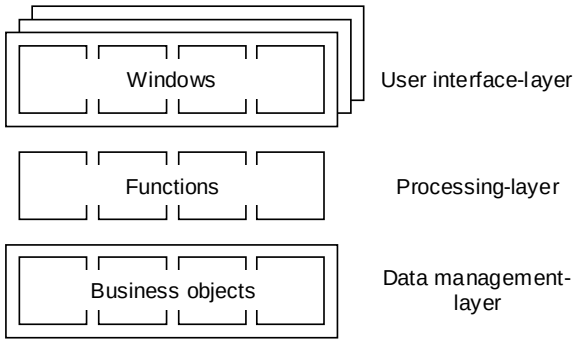


**Figure 3. Units of flexibility**

If we want to use these components in other applications we should make them as independent as possible. The independence of business objects is further increased by creating separate objects for the relations between them. These relation objects are business objects that consist of the key-values of the business objects that participate in the relation.

To make a distinction between components in the processing-layer that do access components of the data management-layer and those who do not, we have divided them into two categories: collaboration components and functional components. Collaboration components could be impacted by changes in the data management-layer, but functional components will not.

Note that we have not yet addressed the question whether these components offer the right domain knowledge. We postpone this question to the evaluation in section 5.6.

The above discussion might suggest that developing a software architecture is a linear process. It is not. It is the result of an iterative process of considering all factors and translating them into a model.

### 5.3. Communication architecture

Distributing the components of the application over a number of clients and servers also raises the question of how the distributed components communicate with each other. They can not access each other directly because they are placed at different locations and they do not know each other's location. In this example we have chosen to use a central message-broker to send messages between the distributed components.

Using a central message-broker puts a number of constraints on the components of the application. First, upon creation a component needs to register with the message-broker, for if the message-broker does not know the existence and location of a component it can not send messages to it. Second, when a component wants to send a message to another component it sends the message to the message-broker which forwards the message to the destination component. This introduces two new problems: how can the destination of a message be specified and how do the components know where the message-broker is. The first of these problems is solved by assigning a unique component-identifier to each component. This identifier consists of a part that indicates the component and a part that indicates the location of the instance of the component. The first part of the identifier is assigned to a component at design-time and the second part is assigned at run-time by the message-broker.

The second problem could be solved by hard-coding the location of the message-broker in each component. The drawback of this approach is that it limits the scalability of the application, for it prohibits us to add another message-broker if the number of users increases. We decided to solve the problem by placing a small part of the message-broker, often called a stub, on each machine. Sending a message from one component to another now results in the following sequence of actions. The component sends the message to the stub of the message-broker, together with the component-identifier of the target component. For all components, except the user interface elements, providing the component class-identifier is sufficient, because based on this identifier the message-broker can locate the target component. However, for user interface elements the location-identifier is also necessary, because multiple instances of the same component might exist. The stub then forwards this message to the central message-broker. The message-broker locates the target component, forwards this message to the stub at the target location, which delivers the message to the target component.

Calling methods by sending messages means that calling the method and receiving an answer are separated. In section 5.4 we indicate how this mechanism is supported by the components. In Figure 4 the communication structure of the application is shown.

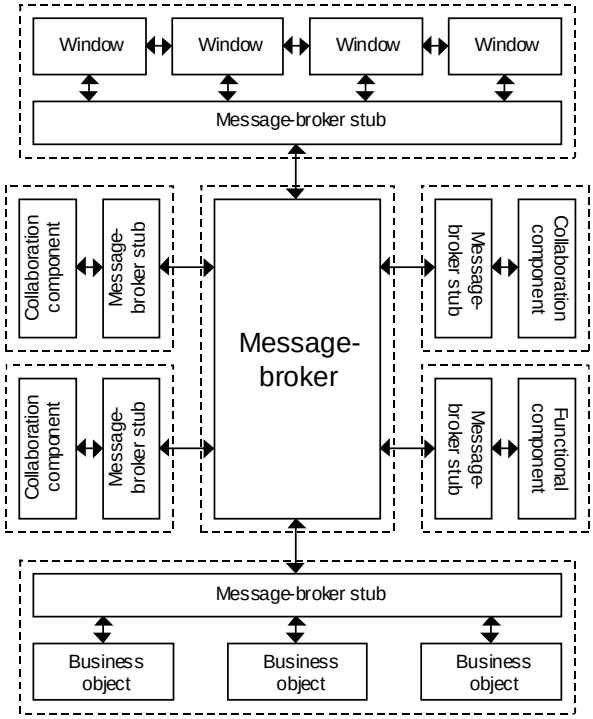


**Figure 4. Communication between the components**

### 5.4. Framework

In this section we describe how a framework for message-brokerage is created. We will assume that the framework will be implemented in Java, but we will not show its full implementation. We focus on three things: (1) a definition of the interface of the service of the framework, (2) a definition of the abstract classes that serve as variation points and (3) a definition of the abstract classes from which the components of our application can be derived. Java does not support multiple inheritance, but provides the notion of interfaces to support a similar mechanism (see [1]). An interface definition can be regarded as a fully abstract class. To distinguish abstract and interface definitions from concrete ones we use the UML convention, which is to italicize the name of abstract definitions and to add the keyword <<interface>> to interface definitions (see [4]).

**5.4.1. Services.** The service our framework offers is message-brokerage. In our architecture we have chosen to use one central message-broker and to place stubs of this message-broker on each location. The application-components call these stubs to send messages to the message-broker and the message-broker uses these stubs to deliver these messages. So, in fact the definition of the stub is the interface of the framework. The class

definition of the message-broker stub (including only the public methods) is shown in Figure 5.

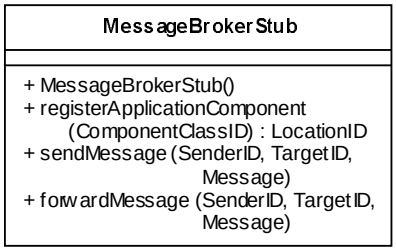


**Figure 5. Class definition of MessageBrokerStub**

Besides its creator, which registers the stub with the central message-broker, the message-broker stub has three public methods. The first is used to register application-components (in section 5.4.3. we explain what an application-component is). The second method is invoked by application-components and takes care of sending a message from one application-component to another. The third method is invoked by the message-broker and forwards messages to components that are located on the machine of the stub.

A ComponentID consists of a ComponentClassID and a LocationID. When a message is sent through the message-broker, the component identifier of the sender and target component are given. All components, except window-components, are unique and can be found based on their ComponentClassID. Thus, supplying this part of the ComponentID is sufficient for these components. Messages to window-components should include the LocationID to indicate the instance of the component that is meant.

**5.4.2. Variation point.** A variation point is a part of the framework for which the implementation is not fixed, but can be chosen to fit the situation at hand. In this framework we have chosen to make the message-broker a variation point. This enables us to change its implementation independently of the rest of the framework. For the framework this means that only the interface definition of the MessageBroker is included. Unless an implementation of this interface is given, the framework will not function. In Figure 6 the definition of the interface is shown.

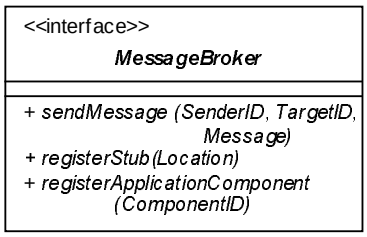


**Figure 6. The interface of MessageBroker**

We can ask ourselves whether the class that provides an implementation for this message-broker is a component. We think it is, because it has all properties of components we mentioned in section 3.1. It is an example of a technical infrastructure component.

**5.4.3. Abstract classes and interfaces.** We have chosen to create one interface for all application-components. An application-component is a component that belongs to the application and not to the framework. By defining this interface, we allow the message-broker stubs to access the application-components independent of their type. The interface definition is given in Figure 7.

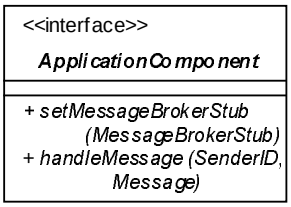


**Figure 7. The interface of ApplicationComponent**

The method *setMessageBrokerStub(MessageBroker-Stub)* informs the component of the message-broker stub it should use. The method *handleMessage(SenderID, Message)* is called when the component receives a message. It should call the requested method and afterwards return the result of this method to the sender.

The business objects of the application have a number of properties and methods in common. These common elements are captured in the abstract class BusinessObject. Its class diagram is shown in Figure 8.

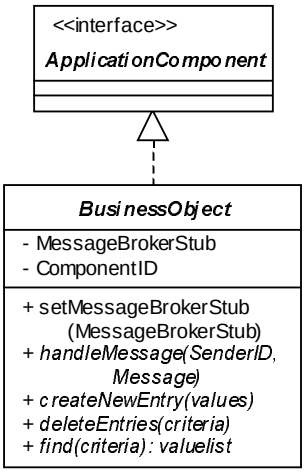


**Figure 8. Class diagram of BusinessObject**

Likewise, we have chosen to create abstract base classes for collaboration components, functional components and window-component. We have omitted their class diagrams, because these diagrams are similar to the class diagram of BusinessObject.

Each of the machines that will be used for the system, needs to be initialized to create a message-broker stub and the application-components that will run on that machine. We have chosen to create two abstract classes, ClientApplication and ServerApplication, from which the initializing classes for clients and servers should be derived. The main difference between them is that after starting the application on a client the main-window is shown. An interesting question is if a descendant of one of these classes is a component. Though we can describe the relations of this class with other parts of the application, we feel it is too tightly integrated with a specific application and cannot be reused in another application. Therefore, we will not consider this class a component. Apparently, if a class is too tightly integrated with a specific application it is not a component.

## 5.5. Components

In the previous sections we have shown from which types of components our application is built. In this section we work out two of these components, a business object and the message-broker.

The business objects are the classes and the relations of the class diagram in Figure 1. We will focus on the Employee business object. Its class hierarchy is shown in Figure 9.

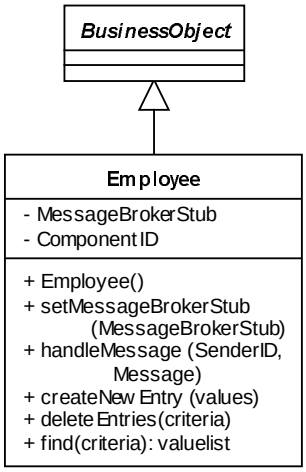


**Figure 9. Class diagram of the Employee business object**

The contract for the Employee business object includes the aspects mentioned in section 3.2, except for the service level. We have omitted the specification of this service level because it is dependent on the implementation of the component and we did not implement the components.

- *Functionality*: The component Employee registers the name and address of the employees of an organization. The component handles the communication with the database management-system for the employee data.
- *Environment*: The Employee component makes a number of assumptions about its environment.
  - *Infrastructure*: The Employee component assumes the presence of a message-broker stub and a Java Virtual Machine (including JDBC-support) in its environment.
  - *Control structure*: The component does not have the primary thread of control. It is only activated when one of its functions is called. The component can handle only one concurrent transaction. When the component executes a request, other requests are stalled.
  - *Data model*: The component expects a database table named Employee with two columns:
    - Name: String[30]
    - Address: String[60]

  *Interfaces*: The component has both an upper- and a lower-interface. The upper-interface consists of the public methods defined in Figure 9. The lower interface consists of the way the component accesses the message-broker and the JDBC-driver. It calls the following functions of the message-broker stub:
  - sendMessage (SenderID: ComponentID, TargetID: ComponentID, Message: Message)
  - registerApplicationComponent (ComponentID: Integer): LocationID

  The JDBC-driver is accessed according to the JDBC API. We will not go into the details of this API.

The message-broker is a variation point of the application. We need a component that implements its interface. The framework hides the complexity of the communication between the message-broker and its stubs. For the component this communication is completely transparent. Its class definition is given in Figure 10.

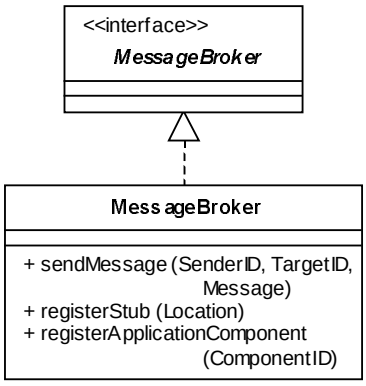


**Figure 10. Class diagram of MessageBroker**

The component has the following contract:

- *Functionality*: The message-broker sends incoming messages to the target location.

- *Environment*: The message-broker makes the following assumptions about its environment.
  - *Infrastructure*: The message-broker assumes the presence of a Java Virtual Machine in its environment.
  - *Control structure*: The message-broker has the primary thread of control. It waits for requests and executes them when they come in. The message-broker can execute only one concurrent request. The environment should queue the requests as they come in.
  - *Data model*: The message-broker maintains a list of application-components and their location and a list of message-broker stubs and their location.
- *Interfaces*: The upper-interface of the message-broker consists of the public functions defined in the class diagram in Figure 10. The lower-interface consists of the following method of the message-broker stubs:
  - forwardMessage (SenderID: ComponentID, TargetID: ComponentID, Message: Message)

## 5.6. Evaluation

In this section we have applied the ideas about components to an example. We showed how the components of the system are determined by making a number of decisions about the software architecture. It is likely that when these decisions are taken differently, different components would be distinguished. For example, when the components of the system are not distributed we would not need components that communicate through a message-broker.

It is useful to assess whether we achieved our goals: reusability, flexibility and reduction of the complexity of the distributed development. The reusability of our components can be judged by determining in which other applications they can be used. This depends on two things:

- For which applications does the component deliver the right functionality?
- In which other environments does the component fit?

To answer these questions we need to make a distinction between IT-components and business components. IT-components do not implement domain-specific functionality and therefore they are usable in different user-domains. However, they are closely related to the architecture of an application. Not only should they fit this architecture, but they should also implement functionality that is relevant for the architecture. This means that for IT-components the first question should state: For which *architectures* does the component deliver the right functionality? Our example included one IT-component, namely the message-broker. This message-broker is possibly reusable in other architectures, as long as message-brokerage is used in this architecture.

Business components, on the other hand, are somewhat different. Their functionality is specific for a user-domain and, as van Vliet ([10]) remarks, they should reflect the primitive notions of the user-domain at the right level of abstraction. We suspect the level of abstraction of our business components to be too low, because the concepts they represent are not the concepts of the user-domain, they are too technical. Research will have to show if we can find components at the right level of abstraction for reuse and are still able to achieve our other goals.

The flexibility of the application can be assessed by considering the effort necessary to implement changes to the requirements. We need to make a distinction between functional and technical flexibility. Functional flexibility concerns changes to the functional requirements of an application and technical flexibility concerns changes to the technical requirements (such as performance and security). During the development of the architecture we took into account the fact that the system will be extended to record more information about employees. We took measures to limit the impact of these changes. An interesting question is if these measures also permit changes to the functionality that were not anticipated. Because our application is structured according to the class diagram, this means that changes that have a great impact on our class diagram also have a great impact on our application.

We did not consider any changes in the technical requirements, but it is interesting to see what happens when these requirements change. For example, let us assume that our application was originally developed for 10 users, but it will now be used by a multinational with branches all over the world. This has a number of consequences for our application. First, for reasons of performance, our architecture with one central message-broker will not be adequate anymore. We will need some other form of message-brokerage, for example using multiple message-brokers. Also for reasons of performance it is decided that the business objects will not be located on a central server, but that they will be distributed over a number of servers. Next, because people in different countries will use the application, the user interface of the application needs to be available in several languages. And finally, for security reasons it is decided that an authorization mechanism will be introduced. This mechanism should allow us to define rules for accessing entries of our business objects. We should, for example, be able to define a rule that states that only a certain group of people can edit salaries above $50,000. In Table 1 we have shown the effect of these changes. A plus sign (+) means that the change is supported by the architecture of our application and a minus sign (–) means it is not.

| Change | Solution |
|---|---|
| Multiple message-brokers (+) | This change can be encapsulated behind the interface of the message-broker stubs. It requires a new message-broker and possibly an adaptation of the message-broker stubs. |
| Distributed business objects (+) | The business objects are already independent of each other, so distributing them requires little effort. |
| Localized user-interface (+) | The application is not dependent on the user interface, which means that the user interface can be changed without affecting the rest of the application. |
| Authorization mechanism (–) | This additional requirement has an impact on many aspects of our application. It will require us to change almost every component of the application. |

**Table 1. The effect of the proposed change**

Apparently, some changes in the technical requirements impact the business components. An interesting question is which architectural choices help us limit this impact.

In the example we explicitly considered the fact that the components of the application are distributed. However, we cannot judge whether the complexity of the distributed deployment decreased, because we did not show how the application is distributed and deployed. We did, however, see that components that are distinguished for distribution differ from the ones distinguished for flexibility and reuse. In this example the components that are distinguished for distribution are larger than the components distinguished for flexibility and reuse, but this does not always have to be the case. We need to gain further insight into the relations between the goals and the components.

## 6. Conclusion

The purpose of this paper is to describe the ideas we encountered during our exploration of the existing views on components. We started this description by listing the main goals we like to achieve by using components: (1) reuse, (2) flexibility and (3) a reduction of the complexity of distributed deployment. These goals will not be achieved automatically, but during the development of the architecture of a system we should be focussed on achieving them. We also note that frameworks are closely related to components. Frameworks implement complex technical or business mechanisms and can be used as an environment for components.

We applied the ideas described in an example. Although this example is somewhat theoretical, it raises a number of questions, like what is the optimal level of abstraction for a reusable component and how does this influence the flexibility and ease of deployment of the component. Our next step will be to put our ideas into practice. We will work out the architecture sketched in our example and compare it with the architecture of similar systems found in industry. This way, we hope to find answers to questions raised, thus deepening our understanding of choices made.


### Acknowledgements

This research is mainly financed by Cap Gemini Netherlands. We are very grateful to Guus van der Stap and Ad Strack van Schijndel of Cap Gemini Netherlands for their ideas and their comments on the initial text. We would also like to thank the representatives of the various organizations that took the time to talk to us and share their ideas.


## 7. References


[1] Gary Cornell and Cay S. Horstmann. *Core Java*. Prentice-Hall, Upper Saddle River, 1996.

[2] Desmond F. D'Souza and Alan C. Wills. *Objects, components and frameworks with UML*. Addison-Wesley, Reading, 1998.

[3] Mohamed E. Fayad and Douglas C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM* 40, 10 (1997), pp. 32-38.

[4] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, 1997

[5] David Garlan, Robert Allen and John Ockerbloom. Architectural Mismatch: Why reuse is so hard. *IEEE Software* 12, 6 (1995), pp. 17-26.

[6] Ivar Jacobson, Martin Griss and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press, New York, 1997.

[7] Ralph E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM* 40, 10 (1997). pp. 39-42.

[8] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Upper Saddle River, 1996.

[9] Guus van der Stap and Ad Strack van Schijndel, *Layered Component Architecture*. Cap Gemini internal, memorandum, 1998.

[10] Hans van Vliet, *Software Engineering: principles and practice*. John Wiley & Sons Ltd., Chichester, 1993.